Dependency Injection

- One of the biggest issues in software systems today is managing the dependencies between objects. If my ProcessOrdersService class is using the OrdersDAO and CustomersDAO classes, it has dependencies on them and, through them, each of *their* dependencies.
- Unmanaged, those dependencies can get out of control without you even noticing.
- If you have ever changed a constructor signature and realized that you have just broken your code in 19 places, or ever tried to instantiate an object only to find that it needs the environment *just so* because of a dependency three levels down, you know the pain that I am describing.
- *Inversion of Control (IoC) means that objects do not create other objects on which they rely to do their work. Instead, they get the objects that they need from an outside source (for example, an xml configuration file).*
- *Dependency Injection (DI) means that this is done without the object intervention, usually by a framework component that passes constructor parameters and set properties.*
- In object-oriented computer programming DI is a technique for supplying an external dependency (i.e. a reference) to a software component - that is, indicating to a part of a program which other parts it can use.

- Say your application has a text editor component and you want to provide spell checking.
- Here we create a dependency between the TextEditor and the SpellChecker.

```
public class TextEditor
{
  private SpellChecker checker;
  public TextEditor()
  {
        checker = new SpellChecker();
  }
}
```

The Problem – Tight Coupling

- The biggest issue with the code is tight coupling between classes.
- In other words the **TextEditor** class depends on the **checker** object. So for any reason SpellChecker class changes, it will lead to change and compiling of 'TextEditor' class also.
- So let's put down problems with this approach:
  - The biggest problem is that TextEditor class controls the creation of checker object.
  - SpellChecker class is directly referenced in the TextEditor class which leads to tight coupling between SpellChecker and TextEditor objects.
  - TextEditor class is aware of the SpellChecker class type.
  - So if we add new SpellChecker types, it will lead to changes in the TextEditor class also as TextEditor class is exposed to the actual SpellChecker implementation.
- So if for any reason the SpellChecker object is not able to create, the whole TextEditor class will fail in the constructor initialization itself.

Solution

- The solution definitely revolves around shifting the object creation control from the TextEditor class to some one else.
- The main problem roots from the TextEditor class creating the checker object.
- If we are able to shift this task/control of object creation from the TextEditor class to some other entity, we have solved our problem.
- In other words, if we are able to invert this control to a third party, we have found our solution.
- So the solution name is IOC (Inversion of Control).

IoC and DI patterns are all about removing dependencies from your code

- *In an IoC scenario we would instead do something like this:*

```
public class TextEditor
{
  private ISpellChecker checker;
  public TextEditor(ISpellChecker checker)
  {
        this.checker = checker;
  }
}
```

- *Now, the client creating the TextEditor class has the control over which SpellChecker implementation to use.*

- *We're injecting the TextEditor with the dependency.*

Dependency Injection (DI)

- Without the concept of dependency injection, a consumer who needs a particular service in order to accomplish a certain task would be responsible for handling the life-cycle (instantiating, opening and closing streams, disposing, etc.) of that service.

- Using the concept of dependency injection, however, the life-cycle of a service is handled by a dependency provider (typically a container) rather than the consumer.

- The consumer would thus only need a reference to an implementation of the service that it needed in order to accomplish the necessary task.

DI involves at least three elements:

- a dependent,
- its dependencies and
- an injector (sometimes referred to as a provider or container).

- The dependent is a consumer that needs to accomplish a task in a computer program. In order to do so, it needs the help of various services (the dependencies) that execute certain sub-tasks.

- The provider is the component that is able to compose the dependent and its dependencies so that they are ready to be used, while also managing these objects' life-cycles.

- This injector may be implemented, for example, as a service locator, an abstract factory, a factory method or a more complex abstraction such as a framework.

Simple Example

- *In this example we created a component that provides a list of movies directed by a particular director*
- *This stunningly useful function is implemented by a single method.*

```
class MovieLister...
{
    public Movie[ ] moviesDirectedBy(String arg)
    {
        List allMovies = finder.findAll();

        for (Iterator it = allMovies.iterator(); it.hasNext();)
        {
            Movie movie = (Movie) it.next();
            if (!movie.getDirector().equals(arg))
            it.remove();
        }
        return (Movie[ ])  allMovies.toArray(new Movie[allMovies.size()]);
    }
}
```

- *The implementation of this function is naive in the extreme, it asks a finder object (which we'll get to in a moment) to return every film it knows about. Then it just hunts through this list to return those directed by a particular director.*
- The real point is this finder object, or particularly how we connect the lister object with a particular finder object.
- The reason why this is interesting is that I want this wonderful moviesDirectedBy method to be completely independent of how all the movies are being stored.
- So all the method does is refer to a finder, and all that finder does is know how to respond to the findAll method.
- We can bring this out by defining an interface for the finder.

```
public interface MovieFinder
{
    List findAll();
}
```

- Now all of this is very well decoupled, but at some point we have to come up with a concrete class to actually come up with the movies.
- In this case we put the code for this in the constructor of my lister class.

```
class MovieLister...
{
 private MovieFinder finder;
 public MovieLister()
 {
  finder = new ColonDelimitedMovieFinder("movies1.txt");
 }
}
```

- *The name of the implementation class comes from the fact that we're getting the list from a colon delimited text file.*

- Now if we’re using this class for just ourselves, this is all fine.

- But what happens when someone is overwhelmed by a desire for this wonderful functionality and would like a copy of this program?

- If they also store their movie listings in a colon delimited text file called "movies1.txt" then everything is wonderful.

If they have a different name for their movies file, then it's easy to put the name of the file in a properties file.

But what if they have a completely different form of storing their movie listing: a SQL database, an XML file, a web service, or just another format of text file?

In this case we need a different class to grab that data.

Now because we've defined a MovieFinder interface, this won't alter the moviesDirectedBy method.

But we still need to have some way to get an instance of the right finder implementation into place.

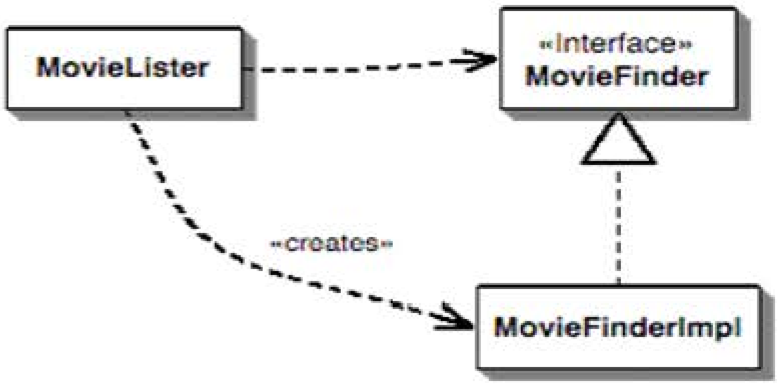


Figure 1: The dependencies using a simple creation in the lister class

Figure 1 shows the dependencies for this situation. The MovieLister class is dependent on both the MovieFinder interface and upon the implementation.

We would prefer it if it were only dependent on the interface, but then how do we make an instance to work with?

This situation could be described as a Plugin.

The implementation class for the finder isn't linked into the program at compile time, since we don't know what others are going to use.

Instead we want the lister to work with any implementation, and for that implementation to be plugged in at some later point, out of my hands.

The problem is how we can make that link so that the lister class is ignorant of the implementation class, but can still talk to an instance to do its work.

Expanding this into a real system, we might have dozens of such services and components.

In each case we can abstract our use of these components by talking to them through an interface (and using an adapter if the component isn't designed with an interface in mind).

But if we wish to deploy this system in different ways, we need to use plugins to handle the interaction with these services so we can use different implementations in different deployments.

- So the core problem is how do we assemble these plugins into an application?

- This is one of the main problems that this new breed of lightweight containers face, and universally they all do it using Inversion of Control.

The basic idea of the Dependency Injection is to have a separate object, an assembler, that populates a field in the lister class with an appropriate implementation for the finder interface, resulting in a dependency diagram along the lines of Figure 2

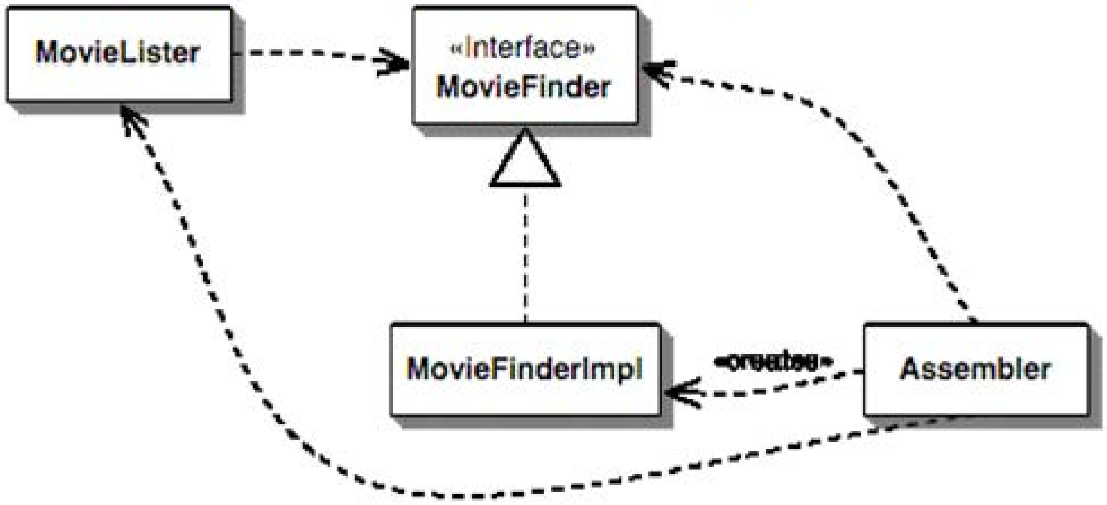
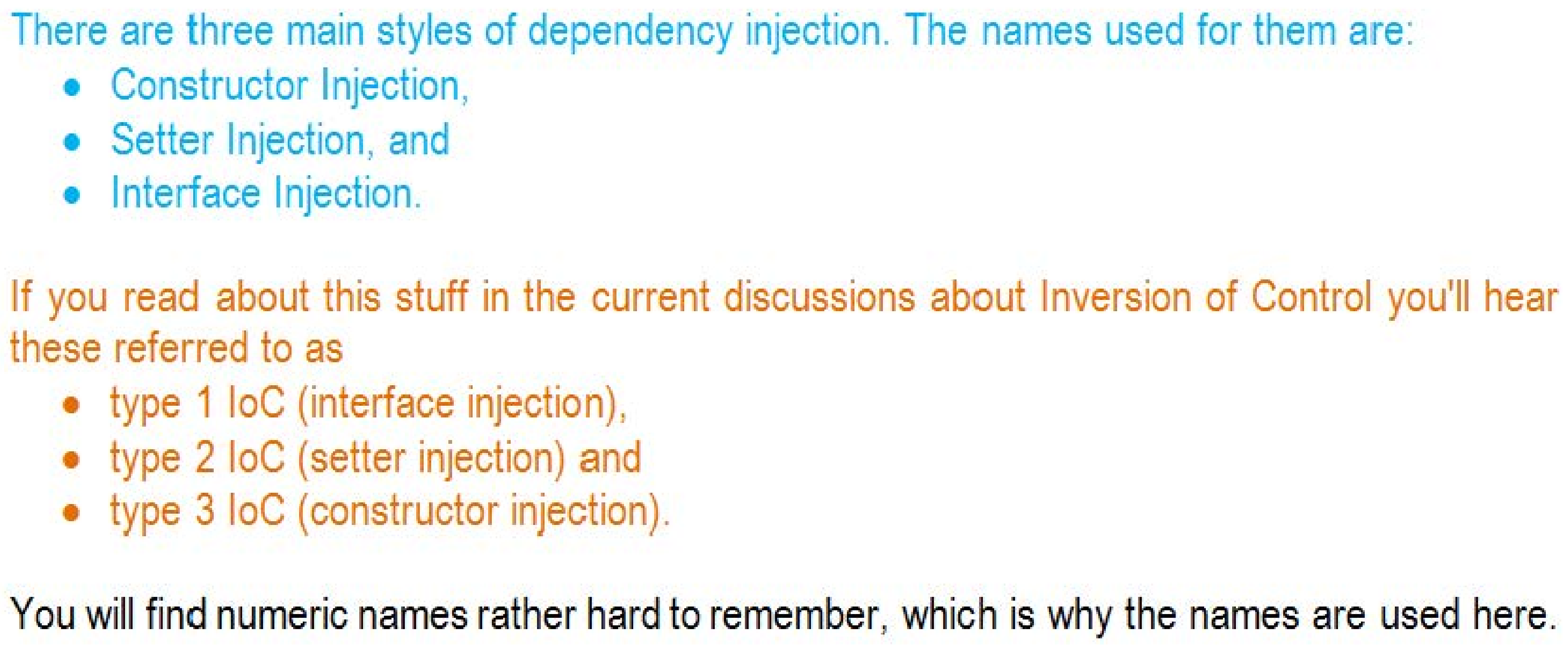


Figure 2: The dependencies for a Dependency Injector

Forms of Dependency Injection


There are three main styles of dependency injection. The names used for them are:

- Constructor Injection,
- Setter Injection, and
- Interface Injection.

If you read about this stuff in the current discussions about Inversion of Control you'll hear these referred to as

- type 1 IoC (interface injection),
- type 2 IoC (setter injection) and
- type 3 IoC (constructor injection).

You will find numeric names rather hard to remember, which is why the names are used here.

Constructor Injection

Let us start with showing how this injection is done using a lightweight container called PicoContainer. PicoContainer uses a constructor to decide how to inject a finder implementation into the lister class. For this to work, the movie lister class needs to declare a constructor that includes everything it needs injected.

```
class MovieLister...
   public MovieLister(MovieFinder finder) {
      this.finder = finder;
   }
```

The finder itself will also be managed by the pico container, and as such will have the filename of the text file injected into it by the container.

```
class ColonMovieFinder...
   public ColonMovieFinder(String filename) {
      this.filename = filename;
   }
```

The pico container then needs to be told which implementation class to associate with each interface, and which string to inject into the finder.

```
private MutablePicoContainer configureContainer() {
   MutablePicoContainer pico = new DefaultPicoContainer();
   Parameter[ ] finderParams =  {new ConstantParameter("movies1.txt")};
   pico.registerComponentImplementation(MovieFinder.class, ColonMovieFinder.class, finderParams);
   pico.registerComponentImplementation(MovieLister.class);
   return pico;
}
```

This configuration code is typically set up in a different class. For our example, everyone who uses our lister might write the appropriate configuration code in some setup class of their own. Of course it's common to hold this kind of configuration information in separate config files. You can write a class to read a config file and set up the container appropriately.

Although PicoContainer doesn't contain this functionality itself, there is a closely related project called NanoContainer that provides the appropriate wrappers to allow you to have XML configuration files. Such a nano container will parse the XML and then configure an underlying pico container. The philosophy of the project is to separate the config file format from the underlying mechanism.

To use the container you write code something like this.

```
public void testWithPico() {
   MutablePicoContainer pico = configureContainer();
   MovieLister lister = (MovieLister) pico.getComponentInstance(MovieLister.class);
   Movie[ ] movies = lister.moviesDirectedBy("Sergio Leone");
   assertEquals("Once Upon a Time in the West", movies[0].getTitle());
}
```

Setter Injection with Spring

To get the movie lister to accept the injection we define a setting method for that service

```
class MovieLister...
private MovieFinder finder;

public void setFinder(MovieFinder finder) {
this.finder = finder;
}
```

Similarly we define a setter for the filename.

```
class ColonMovieFinder...
public void setFilename(String filename) {
this.filename = filename;
}
```

The third step is to set up the configuration for the files. Spring supports configuration through XML files and also through code, but XML is the expected way to do it.

```
<beans>
<bean id="MovieLister" class="spring.MovieLister">
<property name="finder">
<ref local="MovieFinder"/>
</property>
</bean>
<bean id="MovieFinder" class="spring.ColonMovieFinder">
<property name="filename">
<value>movies1.txt</value>
</property>
</bean>
</beans>
```

The test then looks like this.

```
MovieLister lister = (MovieLister) getBean("MovieLister");
Movie[] movies = lister.moviesDirectedBy("Sergio Leone");
assertEquals("Once Upon a Time in the West", movies[0].getTitle());
```

Interface Injection

The third injection technique is to define and use interfaces for the injection. Avalon is an example of a framework that uses this technique in places.

With this technique we begin by defining an interface that we'll use to perform the injection through. Here's the interface for injecting a movie finder into an object.

```
public interface InjectFinder {
    void injectFinder(MovieFinder finder);
}
```

This interface would be defined by whoever provides the MovieFinder interface. It needs to be implemented by any class that wants to use a finder, such as the lister.

```
class MovieLister implements InjectFinder...
    public void injectFinder(MovieFinder finder) {
        this.finder = finder;
    }
```

We may use a similar approach to inject the filename into the finder implementation.

```
public interface InjectFinderFilename {
    void injectFilename (String filename);
}
class ColonMovieFinder implements MovieFinder, InjectFinderFilename......
    public void injectFilename(String filename) {
        this.filename = filename;
    }
```

Then, as usual, we need some configuration code to wire up the implementations.

```
class Tester...
    private Container container;

     private void configureContainer() {
       container = new Container();
       registerComponents();
       registerInjectors();
       container.start();
    }
```

The container way

A container is an abstraction responsible for object management, instantiation and configuration. So you can configure the objects using the container rather than writing client code like factory patterns to implement object management. There are many containers available which can help us manage dependency injection with ease. So rather than writing huge factory codes container identifies the object dependencies and creates and injects them in appropriate objects.

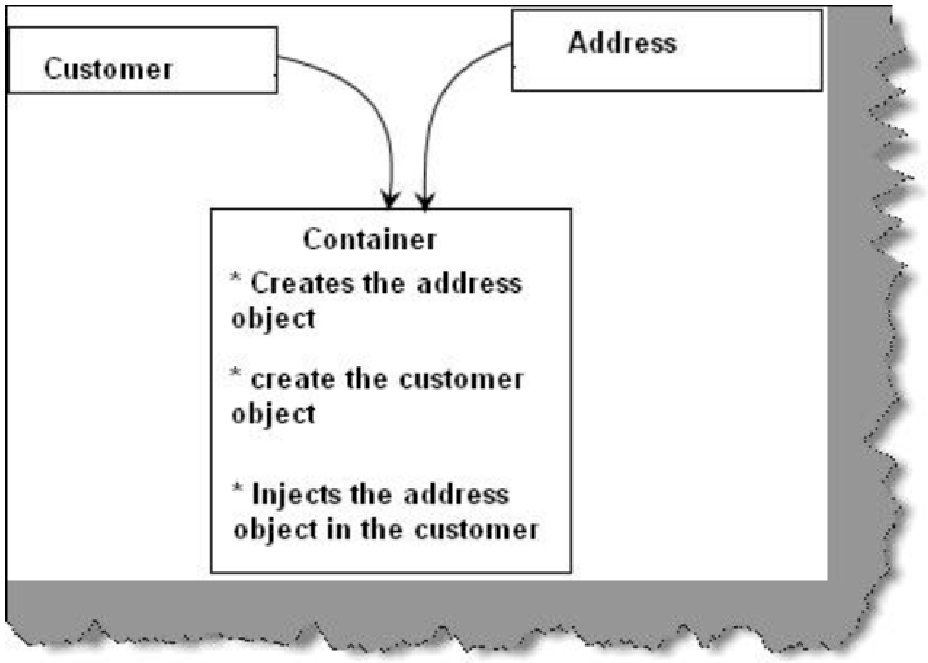

Customer
Address
Container
* Creates the address object
* create the customer object
* Injects the address object in the customer

Figure: - Container in action

So you can think about container as a mid man who will register address and customer objects as separate entity and later the container creates the customer and address object and injects the address object in the customer. So you can visualize the high level of abstraction provided by containers.

What we will do is cover the customer and address example using one of the container Windsor container, you can get more details about the container

Spring IoC Container

- Central to the Spring Framework is its Inversion of Control container, which provides a consistent means of configuring and managing Java objects using callbacks.
- The container is responsible for managing object lifecycles: creating objects, calling initialization methods, and configuring objects by wiring them together.
- Objects created by the container are also called Managed Objects or Beans.
- Typically, the container is configured by loading XML files containing Bean definitions which provide the information required to create the beans.
- Objects can be obtained by means of Dependency injection.
- In many cases it's not necessary to use the container when using other parts of the Spring Framework, although using it will likely make an application easier to configure and customize.
- The Spring container provides a consistent mechanism to configure applications and integrates with almost all Java environments, from small-scale applications to large enterprise applications.
- The container can be turned into a partially-compliant EJB3 container by means of the Pitchfork project. The Spring Framework is criticized by some as not being standards compliant. However, SpringSource doesn't see EJB3 compliance as a major goal, and claims that the Spring Framework and the container allow for more powerful programming models.